MINI LINUX SHELL

## Problem Statement:

Write a mini Linux Shell : having features like running the basic commands such as working pipes(from a single pipe to multiple pipes), the redirection operators ( >, <, etc ), introducing some control characters, remembering the history of the commands.

## Project Objective :

Objective of the project is to build a mini linux shell which will replicate the features of bash shell.

## Introduction:

### What is Shell?

A shell is an interface that allows you to interact with the kernel of an operating system.A Shell provides you with an interface to the Unix system. It gathers input from you and executes programs based on that input. When a program ﬁnishes executing, it displays that program's output.

### How Does a Shell Work?

Every shell has its own language syntax and semantics. In the standard Linux shell, bash, a command line has the form:

***command [arg1] [arg2] ... [argN]***

in which the ﬁrst word is the command to be executed and the remaining words are arguments expected by that command. The number of arguments depends on which command is being executed. For example, the directory listing command may have no arguments-simply by the user's typing “ls” or it may have arguments prefaced by the negative “-“ character, as in “ls –al”, where “a” and “l” are arguments. The command determines the syntax for the arguments, such as which of the arguments may be grouped (as for the “a” and “l” in the “ls” command), which arguments must be preceded by a "-" character, and whether the position of the argument is important.

The command for the command line is usually the name of a ﬁle that contains an executable program, for example, “ls” and “g++” (ﬁles stored in

/bin on most UNIX-style machines). In a few cases, the command is not a ﬁlename but rather a command that is implemented within the shell. For example, “cd” (change directory) is usually implemented within the shell itself rather than in a ﬁle in /bin. Because the vast majority of the commands are implemented in ﬁles, you can think of the command as actually being a ﬁlename in some directory on the machine.

Following steps that a shell must take to accomplish its job

1. *Print a prompt*
2. *Get the command line.*
3. *Parse the command.*
4. *Find the ﬁle.*
5. *Prepare the parameters.*
6. *Execute the command.*

The Bourne shell uses multiple processes to accomplish this by using the UNIX-style system calls fork(), execvp(), and wait().

## Concepts Used:

### I/O Redirection

A process, when created, has three default ﬁle identiﬁers: stdin, stdout, and stderr. These three ﬁle identiﬁers correspond to the C++ objects cin, cout, and cerr. If the process reads from stdin (using cin) then the data that it receives will be directed from the keyboard to the stdin ﬁle descriptor. Similarly, data received from stdout (using cout) and stderr (using cerr) are mapped to the terminal display. The user can redeﬁne stdin or stdout whenever a command is entered. If the user provides a ﬁlename argument to the command and precedes the ﬁlename with a “less than” character "<” then the shell will substitute the designated ﬁle for stdin; this is called redirecting the input from the designated ﬁle.

The user can redirect the output (for the execution of a single command) by preceding a ﬁlename with the right angular brace character, ">" character.

The user can redirect the output (for the execution of a single command) by preceding a ﬁlename with the right angular brace character, ">" character.

### Shell Pipes

The pipe is a common IPC mechanism in Linux and other versions of UNIX. By default,a pipe employs asynchronous send and blocking receive operations.Optionally, the blocking receive operation may be changed to be a non-blocking receiver. Pipes are FIFO (ﬁrst-in/ﬁrst out) buﬀers designed with an API that resembles as closely as possible a low level ﬁle I/O interface. A pipe may contain a system-deﬁned maximum number of bytes at any given time, usually 4KB. A process can send data by writing it into one end of the pipe and another can receive the data by reading the other end of the pipe.

## Methodology:

1. **System calls Used**

## fork()

->

The fork() system call creates a new process that is a copy of the calling process, except that it has its own copy of the memory, its own process ID (with the correct relationships to other processes), and its own pointers to shared kernel entities such as ﬁle descriptors. After fork() has been called,

two processes will execute the next statement after the fork() in their own address spaces: the parent and the child. If the call succeeds, then in the parent process fork() returns the process ID of the newly created child process and in the child process, fork() returns a zero value.

## execvp()

->

The execvp() system call changes the program that a process is currently executing. It has the form:

***execvp(char\* path, char\* argv[]);***

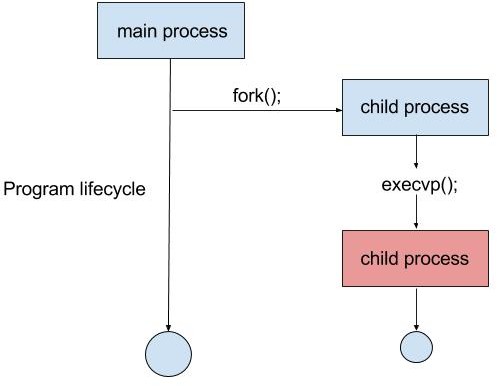
The path argument is the pathname of a ﬁle that contains the new program to be executed. The argv[] array is a list of parameter strings. When a process encounters the execvp() system call, the next instruction it executes will be the one at the entry point of the new executable ﬁle. Thus the kernel performs a considerable amount of work in this system call. It must:

* ﬁnd the new executable ﬁle,
* load the ﬁle into the address space currently being used by the calling process (overwriting and discarding the previous program), - set the argv array and environment variables for the new program execution, and start the process executing at the new program's entry point.

## wait()

->

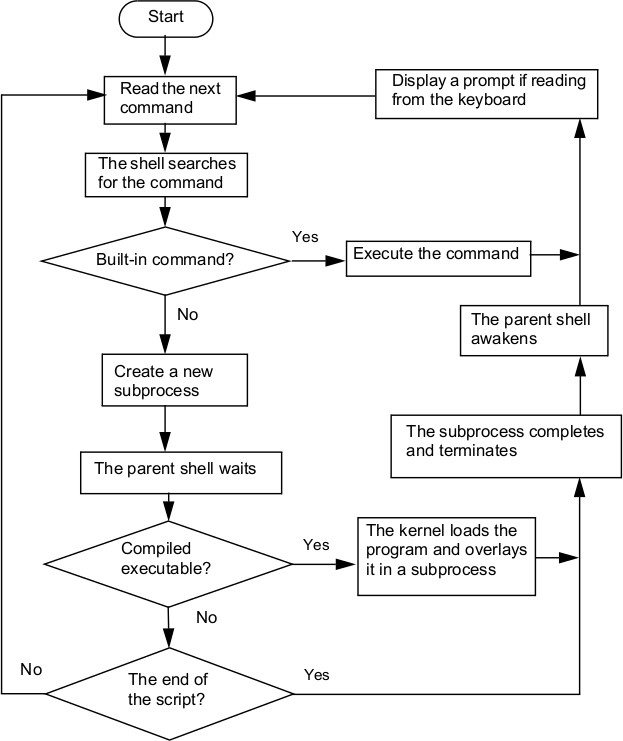
The wait() system call is used by a process to block itself until the kernel signals the process to execute again, for example because one of its child processes has terminated. When the wait() call returns as a result of a child process's terminating, the status of the terminated child is returned as a parameter to the calling process



## Implementation

* 1. **Software Requirements:**
     + Language: C
     + Platform : Linux

### Flowchart



* 1. **Brief description of the working**

1. An inﬁnite while loop is created, printing current directory and waiting for single line user input.
2. If the user calls for cd, the directory is changed to a given path if the path exists.
3. If the user calls setenv with proper format, then a new environment variable is created with speciﬁed value. If an environment variable already exists, then it's value is changed.
4. If user calls printenv, then:
   * If no environment variable is given, all environment variables are printed with their values.
   * Else the values of speciﬁed variables are printed.
5. If the user calls history, then the history ﬁle is printed.
6. If the user types exit/quit/x, the history ﬁle is deleted, all memory is freed, loop is broken and the program terminates.
7. If there is piping in input, after all validity checks, two child processes are created and then both internal and external commands are executed.
8. If there is redirection, after all validity checks, a child process is created and both internal and external commands are executed with proper redirections to the ﬁles.
9. In all other cases, user input is executed using a child process and if anything invalid, an error is displayed.
10. In all these cases, the input is saved in the history ﬁle in sequence.
11. Then all memory is freed and the loop continues from step 1.

### Code:

* 1. **Libraries Included:**

#include <stdio.h> #include <stdlib.h> #include <unistd.h> #include <limits.h> #include <string.h> #include <stdarg.h> #include <fcntl.h>

#include <sys/ioctl.h> #include <sys/types.h> #include <sys/wait.h> #include <sys/stat

* 1. **Define Statements:**

Following are various ***deﬁned*** statements since these are used in our program regularly.

#define print\_error\_message fprintf(stderr,"\033[1;91mERROR: Sorry! No such command exists in my mini shell.\n\033[0m");

#define print\_to\_file fprintf(history\_ptr, "\t%d. %s\n", serial, user\_input);

#define RESET printf("\033[0m"); #define RED printf("\033[1;91m");

#define CLEAR printf("\e[1;1H\e[2J");

* 1. **Created Functions:**

# int count\_argument\_numbers(char\*);

-> This function counts the number of arguments in user input keeping in mind all whitespaces, double quotes, redirection and piping symbols.

# char\*\* find\_all\_paths();

-> This function will ﬁnd all paths that are available in shell to execute the external commands.

# char\*\* separate\_user\_input(char\*, int);

-> This function parse the user input into diﬀerent arrays keeping in mind all whitespaces, double quotes, redirection and piping symbols.

# int is\_present(char\*\*, int, char\*);

-> This function checks whether a particular string is present in the parsed user input

# int find\_positions(char\*\*,int,char\*,int\*\*);

-> This function counts and ﬁnds all accurance of a particular string in the parsed user input.

# char\*\*\* split\_commands(char\*\*, int, int);

-> This function will split the parsed user input into two diﬀerent commands at a given position.

# char\*\* find\_command(char\*\*,int,int);

-> This function ﬁnds the main command after removing symbols and redirected ﬁles.

# char\* get\_program\_path(char\*, char\*\*);

-> This function ﬁnds the path of the external command.

# void print\_message(char\*,char);

-> This function prints a center-aligned message on screen.

# void print\_env\_var\_error(char\*\*);

-> If the user types an environment variable without echo or printenv, then show error and correct form.

# void execute\_cd\_command(char\*\*,int);

-> This function executes the change directory operation.

# void execute\_history(char \*\*);

-> This function executes the history command using cat, i.e., print the history ﬁle.

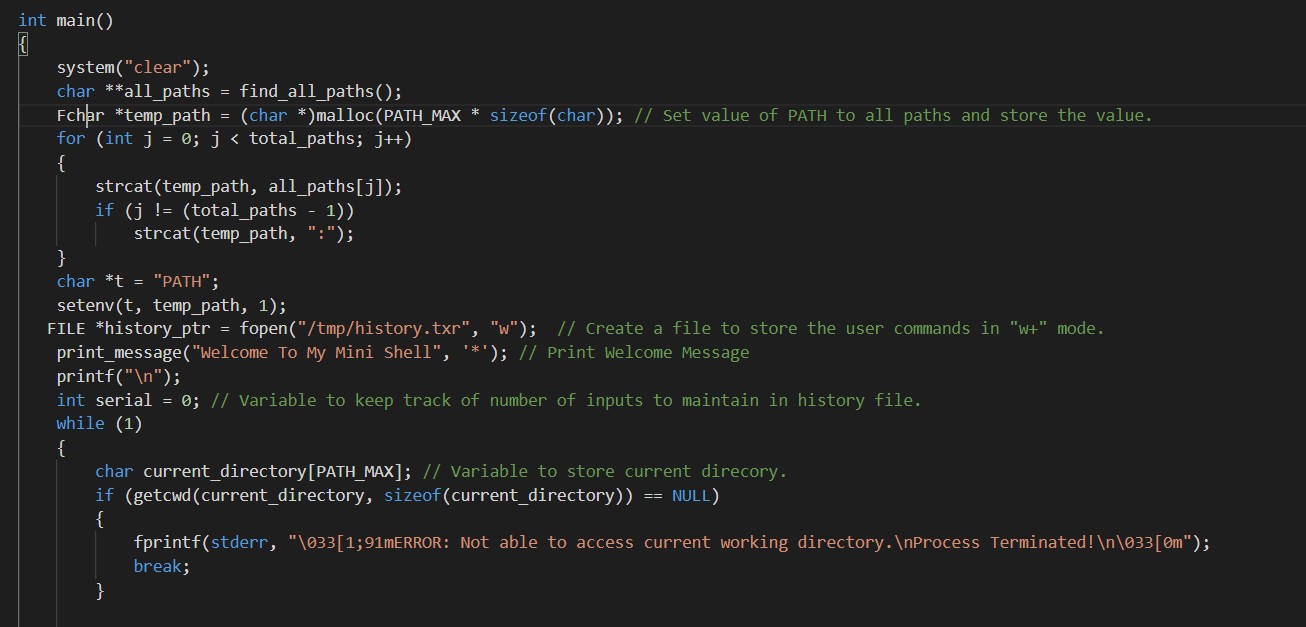
# void execute\_commands(char\*\*,char\*\*,int);

-> This function executes all internal and external commands in the shell.

# char\*\* execute\_env\_var(char\*\*,int,char\*\*);

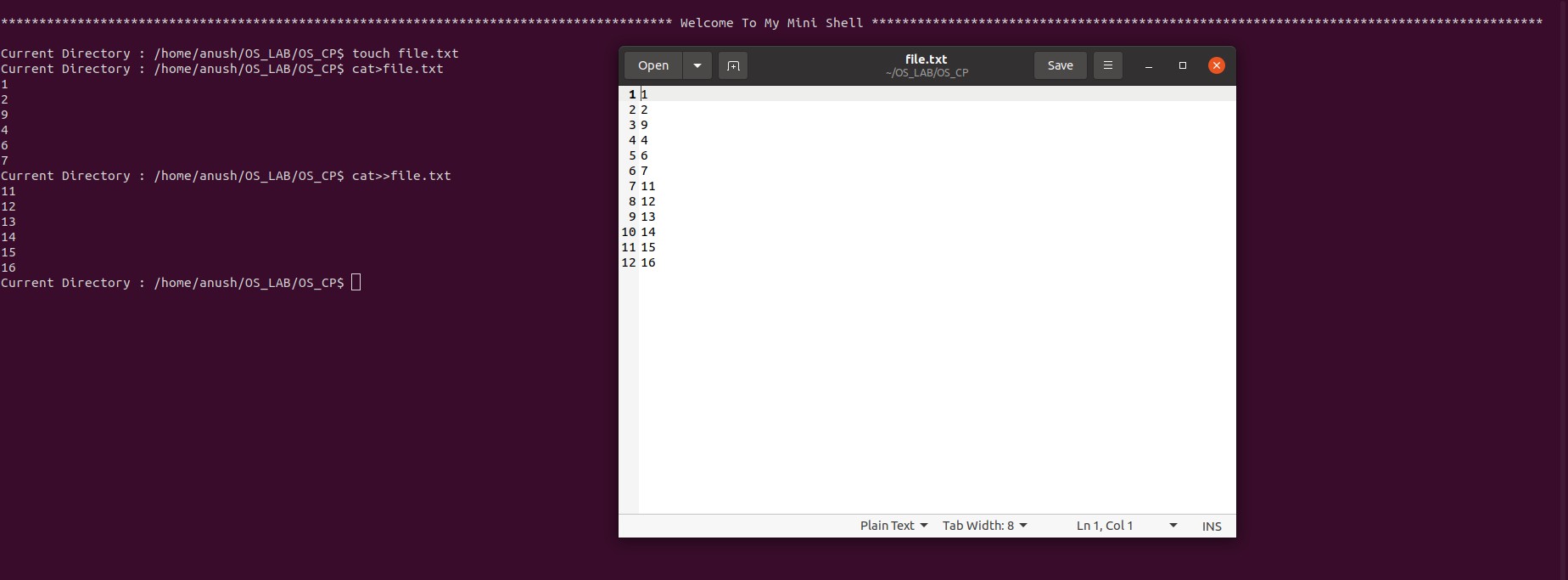
-> This function replaces the value of the environment variable into our input array.

# Main Function



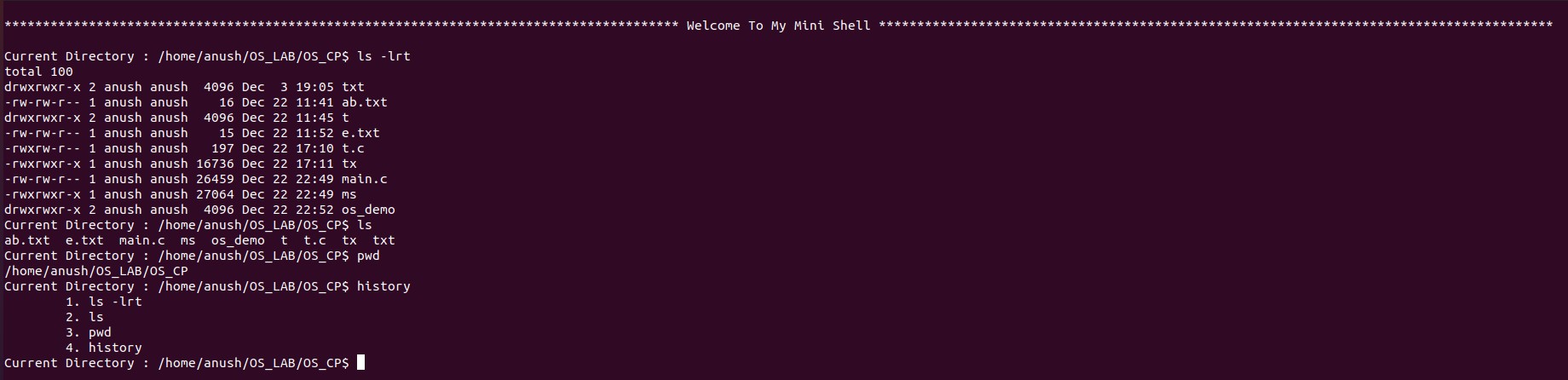
## Output Screenshot:

### Redirection



* + 1. **Pipe**

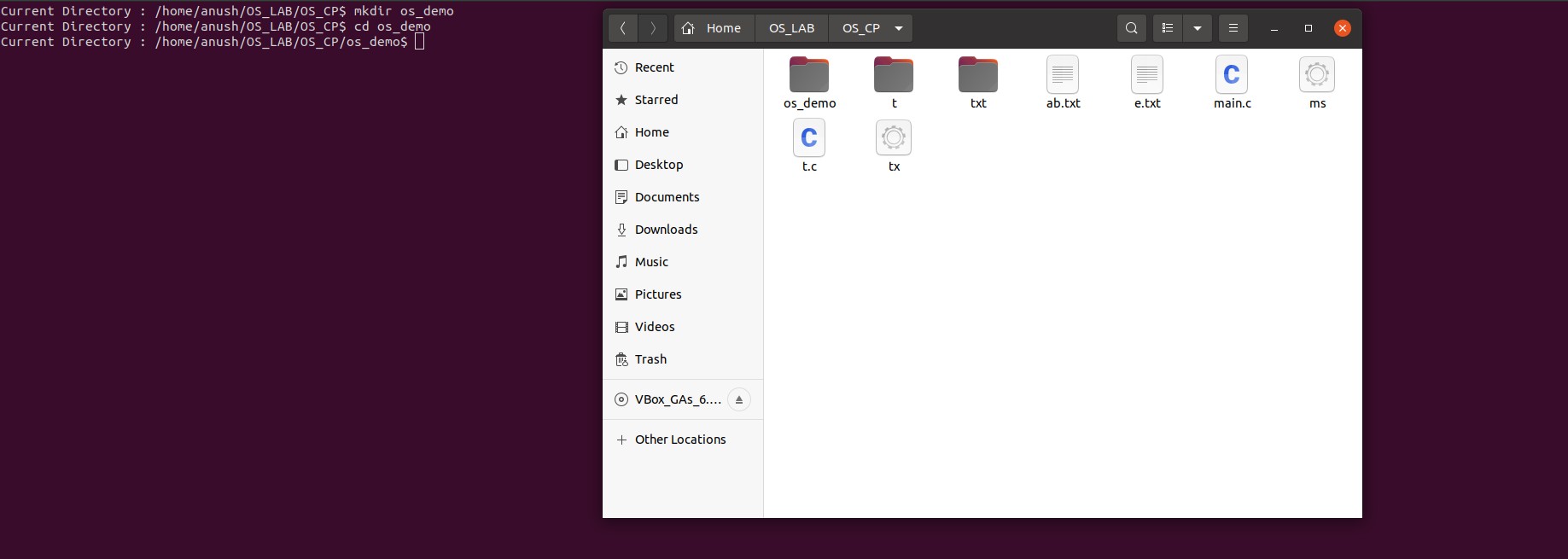
### History Commands



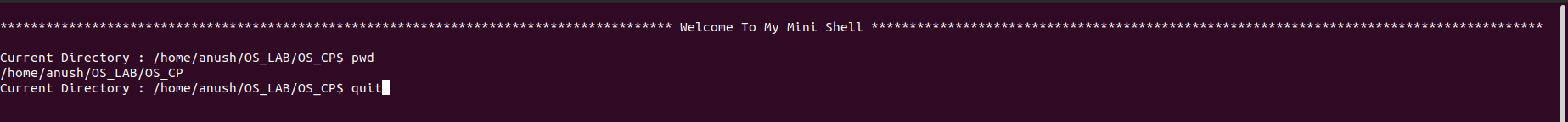
* + 1. **Basic Command**

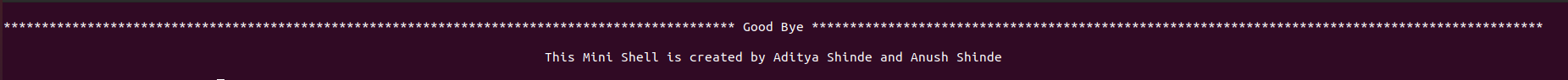


### CD Command



* + 1. **Quit Command**





### Reasonable list of external linux commands

